A Tool for Modeling Concurrent Real-Time Computation

D.D. Sharma, Shie-rei Huang, Rahul Bhatt, N.S. Sridharan

FMC Corp.
Knowledge Systems Research
Artificial Intelligence Center
Central Engineering Laboratory
1205 Coleman Ave., Box 580,
Santa Clara, CA 95052

Review Area: Real-Time Performance

ABSTRACT

Real-time computation is a significant area of research in general, and in AI in particular. The complexity of practical real-time problems demands use of knowledge-based problem solving techniques while satisfying real-time performance constraints. Since the demands of a complex real-time problem cannot be predicted (owing to the dynamic nature of the environment) we need powerful dynamic resource control techniques to monitor and control the performance. In this paper we briefly describe a real-time computation model for a real-time tool, an implementation of the QP-Net simulator on a Symbolics machine, and an implementation on a Butterfly multiprocessor machine.

1. Introduction

Real-time computations is a significant area of research in general, and in AI in particular [1]. The tradition work on real-time has focused on the execution performance of programs and has lead to development of real-time operating systems where the emphasis is on optimizing execution speed of various parts of the operating system kernel. The complexity of practical real-time problems demand use of knowledge-based problem solving techniques while satisfying real-time performance constraints. The complexity of the problem requires that apart from the program execution speed we should also address other performance metrics such as responsiveness, timeliness, and graceful degradation [2]. Thus in addition to the current tools of real-time operating systems we need support for knowledge-based problem solving and user-controllable dynamic resource control. In this paper we describe the architecture and design of such a real-time tool.

Our real-time tool is based on a computation model (called QP-Net) to help the application developer in modeling the real-time problem solution [2]. The computational model supports both the parallel and sequential execution of tasks. The parallel computation model supports both the static and dynamic scheduling of processors to tasks. We have further refined the QP-Net model to provide features important to real-time problems such as asynchronous task execution, performance monitoring, and dynamic resource computation, and dynamic resource scheduling. In this paper we briefly describe QP-Net, the multilayered architecture for the real-time tool, a simple example, an implementation of the QP-Net simulator on a Symbolics machine, and an implementation on a Butterfly multiprocessor machine.

2. QP-Net

QP-Net is a computational model for exploiting concurrency inherent in an application. By exploiting concurrency we mean making concurrent aspects of application solution explicit and providing mechanisms to realize concurrent behavior. The objective of QP-Net model is provide a user with conceptual tools to express concurrent solutions in the ways that matches best with his/her understanding of the problem solution. The motivation being that such an approach will lead to "distributed AI solution architectures" and facilitate both systematic development of concurrent solutions and also help understand "distributed intelligent" architectures in an empirical manner.

Given an application problem the QP-Net based solution consists of three parts:

1. The Solution Model which defines asynchronous application tasks and their interrelationships in terms of a directed communication network.
2. The Execution Model which specifies the temporal order in which tasks are to be executed.
3. The Resource Model which specifies scheduling of finite resources statically or dynamically among the application tasks.

The *solution model* defines the logical execution ordering of the tasks. The interaction/communication between the tasks is explicitly defined by directed network links. The network allows the user in developing conceptual solution models by using network links to indicate "enables" and "restrictions" of interactions between the tasks.

The *execution model* consists of two parts. First, the execution of a task is triggered as a result of execution of a predecessor task defined by the network. A task upon execution passes an "execution token" to the tasks at the other end of the link. The execution of tasks is basically asynchronous which is achieved by associating an "input-buffer" with each task containing arriving execution tokens. The second part of the execution model consists of scheduling task instances. Various prioritization policies can be used to select a task token from the "input-buffer" for execution.

The *resource model* consists of two parts. First, resources can be assigned to tasks in some predefined manner. For example, a part of resources may be *dedicated* to a set of critical tasks and rest of the resources can be shared dynamically by the remaining tasks. The second part of resource allocation model consists of dynamically reallocating the resources.

The QP-Net based application solution model is "richer" than traditional concurrent models. Unlike CSP and Petri-Nets, QP-Net allows asynchronous execution of tasks while allowing synchronous execution as a special behavior. Unlike concurrent object-oriented models such as Actors [3], and Mace [4]. QP-Net makes communication explicit and allows dynamic computation of priorities. Unlike any of the existing models, QP-Net allows dynamic reallocation of resources.

3. QP-Net and Real-Time Applications

The QP-Net approach specially addresses the needs of real-time problems by supporting the following:

- A task queue based solution model which supports asynchronous execution of logically parallel tasks.
- User controllable execution control defined based on application specific knowledge and heuristics in designing the network and task priority and scheduling policies.
- User tune-able performance driven resource re-allocation scheme.
- Development of knowledge-based procedures realized by building RT-Tool on an object-oriented platform.

3.1. Task-Queue Based Solution Model

First, a QP-Net solution model essentially consists of a network of logically parallel tasks. One can view the network connecting an arbitrary number of producer and

consumer tasks. Normally these tasks can be executed asynchronously. A producer or a consumer task can in principle be any arbitrary process as long as it can interact with other processes in two respects: It can receive a job to be done from the predecessor processes in the network and it can send a jog to the successor processes in the network. The feature of asynchronous execution makes it possible to model real-time problems because it enables different processes to execute at different speeds which is necessary to satisfy certain real-time requirements.

The QP-Net approach also defines a specific type of model for each of the producer/consumer node in the network. Each node consists of task queue which contains "prioritized tasks" and a server to execute these tasks. The task queues receive tasks from predecessor nodes and enqueue them using certain enqueing discipline. Thus task queues serve both as buffers (thus enabling asynchronous execution) and also as agendas for scheduled tasks. In QP-Net the task queue and server operations are also considered asynchronous. Thus more than processes of task-queue and server can be active simultaneously. The asynchronous operation enables performing task-queue operations to be responsive to environment demands and server operations to be responsive to task-queue demands.

3.2. Execution Control

In real-time applications a solution is required to be responsive to the data or jobs imposed by the environment. The demands of an environment cannot be directly controlled and in some situations it cannot be anticipated. Given a stream of data or jobs it is necessary to determine the priority in which these tasks should be done. QP-Net model explicitly supports used defined priority mechanisms and thus enables incorporating scheduling algorithms to select real-time tasks for execution.

3.3. Resource Allocation

Real systems have finite resources. Given the criticality of certain tasks it is necessary to allocate certain resources as dedicated and others as shared. As discussed in previous section QP-Net model supports both static and dynamic resource allocation and does specifically to control real-time performance.

3.4. Knowledge-based Procedures

Problem solving in real-time application is knowledge-based. Not only are the domain tasks knowledge-based but the control tasks required for execution control and resource allocation can also be knowledge-based. Nothing in QP-Net model precludes knowledge-based approach. The tool described in the next section facilitates building knowledge-based procedures by providing object-oriented support.

4. Prototyping QP-Net Model

4.1. QP-Net Model Primitives

The QP-Net model for a problem is defined as a network of the following primitive elements:

1. **Tasks:** A Task is a description of an user-defined activity to be performed and may involve local data or other behavior. It involves finite and bounded computation. Tasks are independent, can be executed asynchronously, and any dependence is modeled in terms of network. Execution of a task can have one of the following two effects.: changes to the local state of the task or changes posted to out-going network links.
2. **Task Queues:** It is a queue of tasks and serves two purposes: as a buffer to enable asynchronous execution of tasks, and to enable scheduling of tasks for execution. Four types of primitive task queues are recognized: fifo, priority, time-constrained, and synchronized.
3. **Q-Managers:** A Q-Manager manages task inflow and outflow for a set of priority task queues. Normally, task queues will be elements of a q-manager.

4. **Servers:** A Server attends to a specified q-manager. When free, it requests and receives a task from a q-manager, executes it, and goes back to the q-manager for another task. A server is not a physical processor but a process. Thus the behavior of a server can be defined by the user. Also the design of an application solution in terms of q-managers and servers can be independent of the number of processors.
5. **Task Flow Links:** Task flow links connect a q-manager to a server. The links are directed. The links can be best thought of as a pipe . Normally a server can have only one input link but several output links. In some designs (specially designs for synchronized behavior) more than one input links are possible.

4.2. High-Level Components in QP-Net Model

In this section we describe the operations permissible on a QP-Net element and operations to be performed by a QP-Net model. This will specify the semantics of the model elements.

4.2.1. Tasks

Tasks specify the activity to be performed. There are system tasks and application tasks. Here we will focus on application tasks.

Application tasks are typically generated by a server. Upon generation the execution of tasks is controlled by various factors such as its priority. The activity to be performed is expressed as the body of the task. A task has the following attributes:

1. Trigger-Condition: These are the conditions which should be true for the task to be generated.
2. Precondition: A task may not be ready for execution soon after its generation. Prior to its execution various resources should be available, conditions should exist that do not hinder the execution of task, and conditions for initiating the task execution should be true. The collection of all these conditions are defined as **Precondition**.
3. Body: The body of the task defines the activity to be performed.
4. Priority: The priority of the task is determined in relation to its need to have resources made available to it. It is a dynamically computable quantity.

Real-Time Task: Tasks to which the execution time is of critical importance are called real-time tasks. A real-time task has a timer associated with it. The timer is created at the time of task creation. At any time after that the timer can be queried to find the age of the task.

4.2.2. Task Queues

A task queue is a queue of application tasks. The following operations can be performed on a task queue:

1. Create/Delete a task queue
2. Lock/Unlock a task queue
3. Enqueue/Dequeue a task on a task queue

The following four types of primitive task queues are currently supported:

1. FIFO Task Queue
2. Priority Task Queue: A task is enqueued/inserted on the task queue at a location consistent with its priority. Dequeue of the task queue removes the highest priority task.
3. Time-Constrained Task Queue: A task is enqueued/inserted on the task queue based on its deadline. The task with the tightest deadline is at the head of the task queue.
4. Synchronized task queue: Synchronized task queue dequeues the task only when specified synchronization conditions are met.

4.2.3. Q-Managers

A Q-Manager controls the task inflow and outflow of its tasks queues. It may consist of a number of task queues. A Q-Manager has a unique name. Normally, a Q-Manager will be associated with specific types of tasks such as: update sensor data; execute plans for the operation of system, etc. A Q-Manager has procedures to perform the following:

1. Assign priority to a new task
2. Enqueue the task to appropriate task queue
3. Schedule a task for execution
4. Measure specific performance parameters such as the task queue length, time to deadline, the rate at which tasks have been arriving, and the rate at which tasks have been depleting.
5. Estimate the service needed.
6. Update the priorities of the tasks.
7. Change the queue size (i.e., in essense memory) dynamically to either free-up resources or produce more dynamically stable behavior.

An external object can interface with a Q-Manager in the following ways:

1. Send a new task
2. Request a new task

Priority: One of the features of Q-Manager is to estimate the priority of a task. The priority of a task is intended to be a dynamic quantity rather than being a static quantity assigned at the time of the creation of the task. Various factors influencing the priority are:

Time-Stress or the time between now and the deadline to begin or finish the task.

FIFO: A simple default prioritization

Relative Priority: Since priority drives resource reallocation it is estimated relative to the priorities of existing tasks.

4.2.4. Servers

Servers are the logical processors in QP-Net. Servers are logical processes to keep the program independent of assumptions about any particular processor allocation strategy. This is done to:

1. keep the logical structure of the users program simple;
2. provide maximum flexibility for choosing and experimenting with various processor allocation strategy.

A server requests Q-Manager for a task and then execute the task. A server has the following attributes:

1. A unique name
2. Name of the Q-Manager served.
3. A specification of what the server is supposed to do, i.e., a procedure.
4. Name of Q-Managers or output pipes and rules for determining where the results of executing of a task be enqueued.

4.3. Performance Metering

Performance metering can provide insights into the behavior of a program and help identify program features which can be further optimized. Performance measurement/analysis has been an important area in the design and analysis of operating systems. The analysis usually yields results such as the utilization of processors, the computer system throughput, task queue lengths, and response times. These results can then be used to evaluate or modify computer system designs. In QP-Net we are interested in measuring similar characteristics but our objective is to not simply

to collect benchmarking data but to control the performance characteristics of its application. This leads to both a problem and an opportunity. The problem is that much of the computer systems benchmarking techniques are not applicable. The opportunity exists because our measurements can be heuristic and less precise, the lack of precision is overcome by using a feedback control mechanism. We thus choose the heuristic approaches over formal performance modeling approaches.

In QP-Net the primary objective of performance measurement is to collect data about the programs execution which can be used for on-line control the performance of the program with respect to some predefined standards. Thus the performance metering is used both: (1) To provide comparative data to analyze the performance of various programs with respect to stated specifications, and more importantly (2) to provide data to be used in adjusting the resources to achieve the specifications. The following parameters are currently measured:

1. Latency: Average time lapsed between the the generation of a task and the beginning of its execution. If the current average latency is L, the latency of a new task is l, then new average latency, L_{new} can be given by:

$$L_{new} = f \times l + (1 - f) \times L,$$

   where f is a weighting factor with value between 0 and 1. Similar formula is used in the following to compute the average.
2. Task Queue Size: The number of tasks waiting in the task queue for execution.
3. Granularity: The average size of the task, which is measured by elapsed execution time.
4. Inter-service Interval: Average time between two services at a queue.

The problem of how we take these parameter values and assess the performance status of the system is discussed in the section of Performance Control Strategy.

4.4. Resource Allocation

In QP-Net resource allocation essentially refers to assigning processors to the QP-Net based solution. Let us recall that a QP-Net model consists of network of task queues and servers. A pair of task queue (or q-manager) and server is called a QP-Site. Collection of QP-Sites are called Clusters. Processors can be allocated in the following manner:

1. Allocate processors to q-managers and servers.
2. Allocate processors to each site which in turn reallocates them to q-managers and servers.
3. Allocated processors to clusters which in turn can reallocate them to qp-sites and qp-sites can reallocate them to q-managers and servers.
4. Allocate processors to groups of logically parallel activities, called a module. A module can typically contain several q-managers and servers.

For generality, we will say that processors are allocated to modules where a module can be a cluster, a qp-site, q-manager, server, or an arbitrary composition of activities.

4.4.1. Specifying a Module

Given an application solution as a QP-Net work the user can define execution modules. Processors are allocated to modules. A module can be defined in the following manner:

(define-module Mi (Q1 S1 Q2))

where Q1, Q2 are q-managers and S1 is a server.

4.4.2. Schemes for Assigning Processor to Modules

The processor assignment can be either dedicated in which case a fixed number of processor is allocated to a module, or flexible where the processor can serve one of many groups. Given a set of modules, (M1, M2, M3, M4), how do we assign

processors to the modules? The processor allocation is specified by (Module-ID, Processor-ID). If all processors are identical and any module can be executed on any processor then the allocation can be specified as: (Module-ID, Processor Set) where any processor in the processor set can be used.

```
(define-assignment assign-1 (modules '(M1 M2 M3)
                             number-of-processors 4
                            allocation-strategy uniform-random))

(define-assignment assign-2 (modules '(M4)
                             number-of-processors 2
                            allocation-strategy dedicated))
```

4.4.3. Random Allocation Strategy

The dedicated allocation strategy is simple. Here we describe a random shared allocation strategy.

1. With each module M_i, associate a parameter d_i (called demand parameter). Parameters d_i can change values dynamically.
2. Let $D = d_1 + d_2 + \cdots + d_n$. Out of a total of D executions allocate d_i to M_i. One scheme to allocate is to generate a random number, r, in range $(0, D - 1)$. If $\sum_{j=1}^{j=i-1} d_j \leq r < \sum_{j=1}^{j=i} d_j$, then assign the processor to M_i.

4,5. Performance Control Strategy

The objective of performance control strategy is to compute demand factors d_i for each module to which processors are to be allocated. The need to dynamically compute d_i arises because we want to maintain certain level of real-time performance. For example, suppose we want to keep the latency of tasks in a specific task queue to be less then some specified value l_{ref}. Any given allocation of processors to this task queue will be adequate to assure the desired latency as long as the the task arrival rates do not increase or the service rates do not decrease. In real-time situations both of these things can happen. The task arrival rate depends upon the external environment and can change dynamically. The service rate in a finite resource system depends can change if some other module requires more than its allocated service. We need two types of control:

1. the capability to increase processor allocation to task queue if the demand exceed the current capabilities, and
2. the capability to decrease processor allocation if the demand decreases.

The current approach is to associate a demand factor d with each module. If more resources are needed then d is increased. If less resources are needed then d is decreased. *The specific problem thus is to determine when should we change d and how?*

Our approach to determining d is to formulate the problem as a feedback control problem.

Simple Heuristic Control Schemes

First we discuss simple control schemes based on intuitions of how feedback control systems behave and their adaptation to the problem of performance control. In defining these schemes we are assuming a central scheduler ,as discussed in the previous section.

1. Latency Control

   Let l_{ref} be the desired value of latency, l be the predicted value of latency, d be the current demand factor, *int* be the current inter-service time, q be the number of task in the queue waiting for execution, and Δd be the desired change in d. l

can be estimated as:

$$l = q \times int.$$

Assuming a linear relationship between d and l, we have the following rule:

$$\Delta d = \frac{(l - l_{ref}) \times d}{l_{ref}}.$$

It can be shown that the above heuristic corresponds to controlling a constant task-queue size and the service rate is proportional to demand factor.

The above scheme gives a continuous control of latency, i.e., every time a task is added or removed the demand factor d is computed. In order to limit the overhead of computing d and rescheduling resources we can use a "threshold monitoring scheme" described below.

Let l_1 be the upper limit on latency and l_2 be the lower limit on latency. If l is in range (l_1, l_2), then do nothing; else recompute d. This is a simple form of non-linear control.

2. Controlling Queue Size

Let q_{ref} be the desired queue size, d be the current demand factor, and q the current queue size; then,

if $0.8q_{ref} < q < 1.2q_{ref}$
    do nothing;
else
$$\Delta d = \frac{q - q_{ref}}{q_{ref}}.$$

3. Deadline Control

Let us consider a time-constrained task queue. The head of the queue is a real-time task which has the tightest deadline. Let t_d be the deadline for the task to be completed, t be the current time, g be the average granularity of the tasks in this queue, and t_a is a pre-determined constant denoting the allowance time for scheduling task execution.

If $t_d > t + int + g + t_a$, do not schedule the task; else remove the task and execute it. After the execution, if there is a new task in the head of the queue, new demand factor, d_{new} is computer below. Let $k = \frac{t_d - t - g - t_a}{int}$. If $k \leq 0$, the new task should be executed immediately by the same processor; otherwise set $d_{new} = \frac{d}{k}$

5. Implementations

A simulator has been implemented on Symbolics under Genera 7.2 and a prototype is running on a BBN Butterfly Multiprocessor.

5.1. QP-Net Simulator

The advantage of simulation is to be able to:

1. Simulate different architectures.
2. Provide debugging environment.
3. Observe the execution cycle of different processors.
4. Independent Metering of parameters.
5. Observe operation at different rate of execution.

The simulator was designed with the idea that this would enable the user to quickly evaluate different models. This evaluation requires that the system be easily modifiable

and observable. Together with this, the system should have a reasonable speed of operation. Accuracy of the results could be traded for speed. Hence the simulator does not simulate all the detailed communication and resource parameters. Instead, what was provided is a means to give the average values for each of these parameters. Hence the net effect of these parameters is taken into account. Also the simulation provides an environment to easily debug the application under consideration. Most of the multiprocessor environments lack this capability. Using a simulator makes the task of metering independent of the real time clock. Also monitoring of the different metering parameters is made easy because of having the control of the simulation dedicated to a single processor.

5.2. Butterfly Based Implementation

A prototype of QP-Net has been designed and implemented on the BBN Butterfly Multiprocessor using Butterfly Scheme. The prototype contains four main tool components: queue constructs, performance monitor, resource controller, and dynamic resource allocator. An simple object-oriented programming shell is first implemented on the top of the Butterfly Scheme to facilitate the object-oriented design of the QP-Net.

Basic queue constructs are implemented as object classes so that programmers can inherit them to develop application programs. Four kinds of basic queue constructs have been identified and implemented; they are: the FIFO queue, the priority queue, the time-constrained-queue, and the synchronized queue. Queues are supposed to be filled with "real-time tasks". By real-time task we mean a task that has a private timer associated with it. The timer is an object instance which can be read, reset, stopped and set to run. When a real-time task is created, the timer associated with it is reset and run. A task can be very light-weighted. It may contain only minimal information that the server needs to carry out certain actions.

Real-time tasks cannot be all independent. Especially when tasks are generated by breaking up big tasks, certain order of execution must be followed by the small tasks for correct operations. The synchronized queue construct can be used to synchronize tasks in QP-Net. Tasks are synchronized by producing a tagged data each to a synchronized queue. A synchronized queue accepts two or more tagged data each from a different server. The synchronized queue then only send a complete group of data with the same tag to the server. The synchronization mechanism in a synchronized queue is similar to but more powerful than the data-driven synchronization in Petri Net because data available must have the same tag in order to fire a task in a synchronized queue.

The algorithms used for the implementation of resource controller and the dynamic resource allocator were discussed in Section 7.3 and Section 7.4. The prototype has demonstrated some desired features of a real-time system. More experimental results will be documented in the near future.

6. Conclusion

In this paper we have described the design of a real-time tool based on a new computation model (QP-Net). The computation model's features have been specifically customized to address issues relevant to real-time problems. The real-time tool is currently implemented on a Symbolics Lisp machine and a 16-node Butterfly Multiprocessor.

Based on our work so far we have identified several interesting research issues. We find that real-time applications are highly knowledge rich and complex issues of deadline scheduling and resource allocation can be simplified by appropriate task modeling. For example, we have defined a real-time task whose priority is dynamically computed. Our approach also enables treating time both as a resource and a constraint. We find that modeling a problem solution in terms of logically parallel tasks enables maximum advantage of application inherent concurrency. However, tasks need to be coalesced to control the overhead of resource reallocation. In real-time problems it is also required that task merging should be done based on cost effectiveness of

reduced overhead of resource reallocation (by increasing task size) and the loss of responsiveness (because processors will be busy for longer per task).

We have identified a few issues at the level of operating systems where further work can be beneficial. If the system architecture is distributed then a major problem is to improve processor utilization. This calls for monitoring processor state and then migrating objects accordingly. Another possible approach is to distribute an object over the processors and provide schemes for redirecting messages to the appropriate processor. Apart from the issue of processor utilization we also need to worry about the communication overhead. Again the ongoing research in this area needs to take into account the constraint of responsiveness.

If the system architecture is based on distributed memory then processor utilization issue is solved by using dynamic scheduling of processors. An interesting problem arises because of the overhead of task allocation from a single task queue to multiple processors. The speed-up of task dequeing is limited by the saturation behavior. Thus if we are operating near the saturation point then allocation of additional processors will not lead to desired performance characteristics. The allocation scheme will have to be sensitive to this phenomenon. Or even, better the task queue data structure may be split into two and at the OS level the processor scheduling issue be handled.

References

[1] Stankovic, J. A., *Misconceptions about real-time computing: a serious problem for next-generation systems*, Computer, Vol 21, No. 10, Oct. 1988, pp. 10-19.

[2] Sharma D. D. and Sridharan, N. S., *Knowledge-based real-time control: a parallel processing perspective*, in AAAI-88, Saint Paul, Minnesota, Aug. 1988, pp. 665-670.

[3] Hewitt, C., *Viewing control structures as patterns of passing messages* , Artificial Intelligence, 1977, pp. 323-364.

[4] Gasser, L., Braganza, C. and Herman, N., *Mace: a flexible testbed for distributed AI research*, in M. Huhns ed., *Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Los Altos, CA, pp. 119-152.